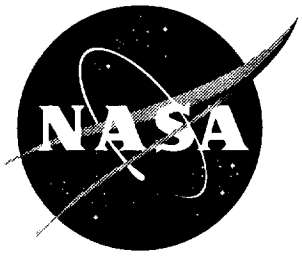
NASA/CR-1998-208434



Flight Guidance System Validation using SPIN

Dimitri Naydich and John Nowakowski
Odyssey Research Associates, Ithaca, NY

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-20335

June 1998

Available from the following:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Abstract

To verify the requirements for the mode control logic of a Flight Guidance System (FGS) we applied SPIN, a widely used software package that supports the formal verification of distributed systems. These requirements, collectively called the FGS specification, were developed at Rockwell Avionics & Communications and expressed in terms of the Consortium Requirements Engineering (CoRE) method. The properties to be verified are the invariants formulated in the FGS specification, along with the standard properties of consistency and completeness. The project had two stages. First, the FGS specification and the properties to be verified were reformulated in PROMELA, the input language of SPIN. This involved a semantics issue, as some constructs of the FGS specification do not have well-defined semantics in CoRE. Then we attempted to verify the requirements' properties using the automatic model checking facilities of SPIN. Due to the large size of the state space of the FGS specification an exhaustive state space analysis with SPIN turned out to be impossible. So we used the supertrace model checking procedure of SPIN that provides for a partial analysis of the state space. During this process, we found some subtle errors in the FGS specification.

Table of Contents

List of Figures

1 Introduction

The mode control logic of the Flight Guidance System (FGS) specified at Rockwell Avionics & Communications [1] is a realistic example of an industrial problem that is compact enough to be a test case for formal design methods. The flight modes determine the mechanisms generating the pitch and roll commands guiding the aircraft. Because of its complexity, an accurate description of the mode control logic is considered a significant problem [2], and it is interesting to see how it can benefit from the application of formal methods.

The mode control logic has been specified using the Consortium Requirements Engineering (CoRE) method [3]. The result is called the FGS specification. The CoRE method supports specifying behavioral system requirements using convenient formal notation with simple and well-defined semantics. A designed system *satisfies* the CoRE requirements if its variables behave in accordance with the requirement specification. As with any formal theory, the general important properties of the FGS specification are consistency and completeness. The *consistency* of requirements means the existence of a system satisfying them. The *completeness* of requirements means that the systems satisfying them exhibit the same variable behavior. Other important required properties are expressed as invariants. An *invariant* is a condition on variable values that should hold for any system satisfying the requirements at any time. Finally, we search for *unreachable* mode transitions, which correspond to code in our executable specification that is never executed.

Although the CoRE semantics is well defined, CoRE does not provide a tool for verifying the properties mentioned above. It is interesting, therefore, to reformulate the FGS specification in another specification formalism that does have mechanical validation support. We have applied SPIN [6,8], a widely used software package supporting the formal verification of distributed systems, to the validation of the mode control logic requirements. To do this, we had to reformulate the FGS specification and the required properties in PROMELA, the input language of SPIN. The reformulation involved a semantics issue, as some constructs of the FGS specification do not have well-defined semantics in CoRE. Then we used the automatic model-checking facility of SPIN, either to validate the required properties or to generate the simulation traces violating them.

The state space of the FGS specification is large. The specification contains about 30 input variables, some of them with thousands of possible values, and about 60 internal state-holding variables related by complex control dependencies. To make model checking feasible, we abstracted away certain irrelevant state information.

The report consists of six parts. Chapter 2 presents an overview of the FGS specification. Chapters 3 and 4 describe the translation of the FGS specification to PROMELA. We describe the representation of the FGS specification properties to be verified in Chapter 5. In Chapter 6, we describe the state space reduction techniques we used to make model checking feasible. We present the validation results obtained with SPIN in Chapter 7.

2 FGS Specification: An Overview

The FGS specification is written in an informal extension of CoRE. The standard CoRE features of the FGS specification are presented in Section 2.1. The extension features are presented in Section 2.2.

2.1 Standard CoRE Features

The CoRE *behavioral model* of a system specifies system variables as functions of continuous time. A specified system is generally considered as a set of mutually interacting finite state machines triggered by events; the *events* in CoRE track changes of expression values in time. The CoRE *class model* structures the behavior model in an object-oriented way.

The system's state information is stored in its internal variables. Some variables like *modes* store their previous state values as well. In general, any variable defined by an event table (see Section 2.1.1) holds some state information. The FGS is very complex. There are about 30 mode variables in the FGS specification, representing flight director modes, lateral flight modes, and vertical flight modes; and there are roughly 30 other internal variables. There are 30 input variables, including both binary variables and variables, like the flight altitude, with thousands of possible values. The system variables are related by complicated control logic.

2.1.1 Tables

Tables are commonly used to represent variable behavior functions in CoRE. There are three types of tables: selector tables, condition tables and event tables. A *selector table* is a tabular representation of strictly mode-dependent information. For example, consider variable **con_HDG_Switch_Lamp,** controlling an indicator lamp. The selector table defining its value is shown in Figure 1. According to the table, the value of **con_HDG_Switch_Lamp** is **Lit** if **mode_Flight_Director** is **ON**, and **mode_Active_Vertical** is **HDG**; it is **Unlit** under all the other possible combinations of the mode values.

Modes		
Mode_Flight_Director	**Mode_Active_Vertical**	**Con_HDG_Switch_Lamp**
OFF	N/A	Unlit
ON	HDG	Lit
	ROLL NAV APPR GA	Unlit

Figure 1: A selector table

A *condition table* represents a function of the mode variables and a set of mutually exclusive conditions. For example, consider variable **con_AP_Engage_Switch_Lamp,** controlling another indicator lamp. The condition table defining its value is shown in Figure 2. According to this table, the value of **con_AP_Engage_Switch_Lamp** in any mode is **Unlit** if **term_AP_Engaged** is **False**; it is **Lit** if **term_AP_Engaged** is **True**.

Mode	Conditions	
All Modes	NOT term_AP_Engaged	term_AP_Engaged
con_AP_Engage_Switch_Lamp	Unlit	Lit

Figure 2: A condition table

An *event table* represents a function that is updated only when an event occurs. Event tables are used to specify variables whose values depend on the system's history. For example, consider variable **term_Selected_Heading,** representing the heading set by rotation of the heading knob. The event table defining its value is shown in Figure 3. According to this table, **term_Selected_Heading** assumes a new value, which depends on its current value, whenever the event **@HDG_Knob_Changed** occurs.

Mode	Events
All Modes	@HDG_Knob_Changed
term_Selected_Heading	MOD(term_Selected_Heading' + 1 degree * term_HDG_Knob_Rotation, 360 degrees)

Figure 3: An event table

A *mode transition table* is a special form of an event table that specifies the behavior of a mode variable having a finite value range. For example, consider submode **ENGAGED** of mode **mode_Autopilot**. The mode transition table defining its value is shown in Figure 4. It specifies that the submode's value changes from **Normal** to **Sync** only when **term_SYNC** becomes true; it changes back when **term_SYNC** becomes false.

Id	From	Events	To
9	Normal	@T(term_SYNC)	Sync
10	Sync	@F(term_SYNC)	Normal

Figure 4: A mode transition table

2.1.2 Partial functions

Tables in CoRE represent total functions. To represent a partial function in CoRE, the specification uses a total "value" function, along with a *sustaining condition* that specifies the domain of the partial function. The value of a partial function is undefined when the sustaining condition does not hold. When the sustaining condition holds, the partial function is equal to the value function, except for the moments when the sustaining condition becomes true. Sustaining condition are provided with *initial values*. At the moment the sustaining condition becomes true, the value of the partial function is set to the initial value. For example, the initial value and sustaining condition of **mode_Active_Lateral** is shown in Figure 5.

Initial Value:	ROLL
Sustaining Condition:	mode_Flight_Director = ON

Figure 5: Initial value and sustaining condition of ***mode_Active_Lateral***

Figure 6 shows the initial value and sustaining condition of **con_HDG_Switch_Lamp. "None"** means that the sustaining condition is tautologically true; "**value function**" means the function specified by the selector table, Figure 1.

Initial Value:	see value function
Sustaining Condition:	none

Figure 6: Default initial value and sustaining condition configuration

2.1.3 Invariants

The invariants imposed on the FGS specification are listed in Figure 7. For example, invariant INV-1 is as follows: **Mode_Active_Lateral = GA ⇒ mode_Autopilot = DISENGAGED**. Thus, the FGS specification asserts that the *Autopilot* must always be disengaged when the *Active Lateral* mode machine is in *Go Around* mode.

INV-1	mode_Active_Lateral = GA ⇒ mode_Autopilot = DISENGAGED
INV-2	mode_Active_Vertical = GA ⇒ mode_Autopilot = DISENGAGED
INV-3	term_AP_Engaged ⇒ mode_Flight_Director = ON
INV-4	(mode_Active_Lateral = ROLL ∧ mon_On_Ground) ⇒ mode_Active_Lateral = ROLL/Hdg_Hold
INV-5	mode_Active_Vertical = GA ⇒ mode_Active_Lateral = GA
INV-6	mode_Active_Lateral = NAV/Track ⇒ term_Selected_Nav_Type ∈ {VOR, LOC, FMS}
INV-7	mode_Active_Lateral = APPR/Track ⇒ term_Selected_Nav_Type ∈ {LOC, FMS}
INV-8	mode_Altitude_Select = CLEARED ⇔ mode_Active_Vertical ∈ {APPR, GA, ALTHOLD}
INV-9	mode_Altitude_Select = ACTIVE ⇔ mode_Active_Vertical = ALTSEL
INV-10	mode_Vertical_Approach = TRACK ⇔ mode_Active_Vertical = APPR
INV-11	term_Overspeed ⇒ mode_Active_Vertical ∈ {ALTSEL, ALTHOLD, APPR, FLC/Overspeed}
INV-12	mode_Active_Lateral = GA ⇒ mode_Active_Vertical = GA

Figure 7: Invariants of the FGS Specification

2.2 Non-Standard Features

To model flight mode logic, the developers of the FGS specification relied on three concepts that have no analogues in the CoRE framework: event cascading, continuous events, and partially defined internal variables. The developers of the FGS specification extended the CoRE notation to express these concepts, and described the intended semantics informally.

2.2.1 Event Cascading

Event cascading defines a conceptual sequencing of events that are simultaneous in real time. Consider the lateral and vertical mode transition tables shown in Figure 8 and Figure 9.

Id	From	Events	To
20	HDG	@HDG_Switch_Pressed	ROLL
21	NAV	@NAV_Switch_Pressed	ROLL
22	NAV	@Nav_Source_Change	ROLL
23	APPR	@APPR_Switch_Pressed	ROLL
24	APPR	@Nav_Source_Change	ROLL
25	GA	@T(term_AP_Engaged)	ROLL
26	GA	@T(term_SYNC)	ROLL
27	GA	@F(mode_Active_Vertical = GA)	ROLL

28	HDG	@HDG_Switch_Pressed	HDG
29	NAV	@NAV_Switch_Pressed	NAV
30	APPR	@APPR_Switch_Pressed	APPR
31	GA	@GA_Pressed	GA

Figure 8: Active lateral mode transition table

Id	From	Events	To
41	GA	@T(term_SYNC)	PITCH
42	VS OR APPR OR ALTSEL OR PITCH	@VS_Pitch_Wheel_Changed	PITCH
43	ALTSEL	@T(mode_Altitude_Select = ACTIVE)	ALTSEL
44	ALTSEL	@CHANGED(term_Preselected_Altitude) WHEN mode_Altitude_Select = ACTIVE/Capture	PITCH
45	ALTSEL	@CHANGED(term_Preselected_Altitude) WHEN mode_Altitude_Select = ACTIVE/Track	ALTHOLD
46	APPR OR ALTHOLD	@ALT_Switch_Pressed	ALTHOLD
47	ALTHOLD	@ALT_Switch_Pressed	PITCH
48	APPR OR VS	@VS_Switch_Pressed	VS
49	VS	@VS_Switch_Pressed	PITCH
50	APPR OR FLC	@FLC_Switch_Pressed	FLC
51	FLC	@FLC_Switch_Pressed	PITCH
52	ALTSEL OR ALTHOLD OR APPR OR FLC	CONTINUOUSLY WHEN term_Overspeed	FLC
53	APPR	@T(mode_Vertical_Approach = TRACK)	APPR
54	APPR	@F(mode_Vertical_Approach = TRACK) AND NOT @GA_Pressed	PITCH
55	GA	@GA_Pressed	GA
56	GA	@T(term_AP_Engaged)	PITCH
57	GA	@F(mode_Active_Lateral = GA)	PITCH

Figure 9: Active vertical mode transition table

Suppose the values of **mode_Active_Lateral** and **mode_Active_Vertical** at time *t* are equal to **GA**. Let external event **@HDG_Switch_Pressed** happen at this time. Then transition 28, Figure 8, instantly changes the value of **mode_Active_Lateral** to **HDG**. This transition invokes internal event **@F(mode_Active_Lateral = GA)** that triggers transition 57, Figure 9, to change the value of **mode_Active_Vertical** to **PITCH**. Analogously, let external event **@VS_Pitch_Wheel_Changed** happen at time *t*. Then transition 42, Figure 9, changes the value of **mode_Active_Vertical** from **GA** to **PITCH**. This invokes internal event **@F(mode_Active_Vertical = GA)** that triggers transition 27, Figure 8, to change the value of **mode_Active_Lateral** to **ROLL**. Transitions 27 and 57 are introduced to comply with the invariants INV-5 and INV-12, Figure 7.

Within the transition cascading model it is important to distinguish the causal relations between simultaneous events. This gives us a criterion to disambiguate mode transition tables with respect to simultaneous events in a causal sequence. CoRE has no such mechanism, and, as a result, the Active Lateral and Vertical Mode Transition Tables are inconsistent in CoRE. To show the inconsistency, we proceed with the sequence of internal events caused by **@HDG_Switch_Pressed**. **mode_Active_Vertical** going to **PITCH** further invokes internal event **@F(mode_Active_Vertical = GA)** to happen simultaneously with **@HDG_Switch_Pressed**. Taking into consideration that **mode_Active_Lateral** is **GA** at that time, we get transition 27 triggered simultaneously with transition 28. Thus we get inconsistency within the CoRE interpretation. However, within the transition cascading model, transition 27 cannot happen because **@F(mode_Active_Vertical = GA)** occurs causally later than **@HDG_Switch_Pressed** , and **mode_Active_Lateral** is already **HDG** by then.

The developers of the FGS specification are aware that CoRE semantics does not permit transition cascading. However, they do not formalize their transition cascading semantics, which makes our problem of formal analysis not completely defined. We define a transition cascading mechanism for CoRE event tables in Section 4.2, similar to the one used in RSML [9], VHDL [10], or Verilog [11] formalisms. A reasonable alternative would be to specify the flight mode logic in one of these formalisms, as they are in some ways more adequate to the problem domain, and are equipped with advanced analysis tools, e.g., [4, 7, 5, 12].

2.2.2 Continuous Events

Transition 52, Figure 9, is triggered by a continuous event, another non-standard feature of the FGS specification. It means that the transition from the source to the target takes place whenever condition **term_Overspeed** is true. The semantics of continuous events is very straightforward in our event cascading model (see Section 3.4.1)

2.2.3 Sustaining Conditions For Modes

In CoRE, sustaining conditions are considered just for controlled, output variables. The purpose of such conditions is to detect when the values of output variables are trustworthy. Sustaining conditions for internal variables like **mode_Active_Lateral** presented in Figure 5 are not legal in CoRE. In general, considering undefined values of internal variables raises the problem of evaluating expressions over such variables. For example, given an integer variable x, what is the value of expression $x \geq 0 \vee x < 0$ when x is undefined? Fortunately, the potentially undefined internal variables in the FGS specification are evaluated in domains that make a simple semantics of undefined values possible (see Section 3.3).

3 Translating FGS specification to PROMELA: Translation Primitives

3.1 Translation Outline

PROMELA is the input language of SPIN. Since our goal is to apply SPIN to the analysis of the FGS specification, we have to reformulate the specification in PROMELA. PROMELA is meant for specifying protocols, which are communication rules for entities exchanging messages over point-to-point channels. A basic event in the PROMELA model is sending/receiving a message.

The problem domain of PROMELA is very different from the FGS (and CoRE) problem domain, so we have to model all the concepts involved from scratch. We use the term *signals* for CoRE monitored variables, controlled variables, modes, and terms in order to distinguish them from PROMELA variables. We represent the concept of event—the change of a signal value in continuous time—by introducing two variables, to hold its current and previous values. For example, Figure 10 shows a CoRE definition of an event identifier. We represent the identified event **@T(mon_HDG_Switch = ON)** as the PROMELA expression

```
mon_HDG_Switch[current] && (! mon_HDG_Switch[previous]),
```

where mon_HDG_Switch is a two-element array indexed by the constants current and previous.

@HDG_Switch_Pressed: event ≡ @T(mon_HDG_Switch = ON)

Figure 10: Definition of an event identifier

Event broadcasting is the mechanism for triggering the flight mode transitions in the distributed CoRE specification. Formalizing event broadcasting in PROMELA requires a complicated system of process synchronization (see the Appendix). Such an explicit synchronization would have to be modified whenever the CoRE specification was modified, and could become a source of additional errors. To avoid this problem, we translated the distributed CoRE specification into a single initial PROMELA process.

The signals in the FGS specification are supposed to be concurrently updated, and they are mutually dependent in general. To simulate concurrent updating of mutually dependent signals by sequential PROMELA code, we introduce an extra variable for each signal to hold its new value. The new value of such a signal depends at most on the current and previous values of other signals. When all the new values for mutually dependent signals are calculated, these signals are updated. Thus, the sequential order of calculating new values and updating mutually dependent signals becomes irrelevant, which adequately simulates the desired concurrency. For example, consider the following CoRE definition of an internal signal, shown in Figure 11.

term_Overspeed: boolean ≡ mode_Overspeed = TOO_FAST

*Figure 11: Definition of **term_Overspeed***

The PROMELA code that computes the new value for this signal is as follows:

```
term_Overspeed[new] = mode_Overspeed[current]== TOO_FAST.
```

The resulting target code quite resembles a C program, as PROMELA has adopted a C-like syntax. Our translation makes no use of the PROMELA constructs that model communicating processes. The only construct used in the translation that has no semantics in C is the *non-deterministic if-statement*. The *restricted form* of non-deterministic if-statement that we use has the following syntax:

```
if
:: condition_1 -> statement_1
        ...
:: condition_n -> statement_n
:: else        -> statement
fi
```

To execute this statement, the conditions are evaluated first, and then some statement following a valid condition is non-deterministically chosen and executed. The statement corresponding to the else branch is executed if none of the conditions is valid. We omit a condition if it is trivially true. We omit the else branch if we believe that one of the conditions is true each time the if-statement is executed.

3.2 Signal Declarations

Regarding the signal representation in PROMELA, we distinguish *event-observable* signals and expressions from the other signals and expressions used in the FGS specification. An *event-observable expression* is a CoRE expression *expr* that either occurs in an event expression @T(*expr*), @F(*expr*), or CHANGED(*expr*), or is a sustaining condition expression used in the specification. For example, consider the definition of **@FD_Pressed**, shown in Figure 12.

@FD_Pressed: event ≡ @T(mon_FD_Switch<left> = ON or mon_FD_Switch<right> = ON)

Figure 12: Example of an event-observable expression

The expression **mon_FD_Switch<left> = ON or mon_FD_Switch<right> = ON** is event-observable. Expression **mode_Flight_Director = ON** is a sustaining condition of **mode_Active_Lateral** (see Figure 5) and other modes, and, therefore, it is event-observable too.

An *event-observable signal* is a signal occurring in an event-observable expression depending just on this signal. For example, signal **term_SYNC** is event-observable because event **@T(term_SYNC)** triggers transition 26 (see Figure 8). On the other hand, input signal **mon_Indicated_Airspeed**, has no event-observable in the specification.

We also distinguish between input, output and internal signals. *Input* signals are those not assigned to in the specification; *output* signals are those not used in computing signal values; and all the other signals are *internal*. All the monitored variables of the FGS specification are input signals; almost all the controlled variables are output signals (see Section 3.2.3); the mode and term variables are internal signals. Signals within each of these groups are updated concurrently. The updating of input or output signals is easily represented by sequential code since neither class of signals contains mutual dependencies. Internal signals are mutually dependent in general. To simulate concurrent updating of mutually dependent signals by sequential code, we introduce an extra variable for each signal to hold its new value, as discussed in Section 3.1.

Signal declarations in PROMELA are defined according to the signal classification above:

1. We declare an input event-observable signal as a two-element array that holds its previous and current values.

2. We declare the other external signals as plain (i.e., non-array) variables.

3. We declare an internal event-observable signal as a three-element array that holds its previous, current, and new value.

4. We declare the other internal signals as two-element arrays that hold their current and new values. We declare an output signal as a regular variable.

We introduce a new two-element array identifier for every non-primitive event-observable expression, to hold its previous and current values. Such an identifier does not correspond to a physical signal, but rather is a shorthand notation making the calculation of complex expression events easier. We consider event identifiers of the FGS specification as shorthand too, and we declare them as plain binary variables. We also introduce auxiliary variables for counting the duration of staying in a particular state. The rest of this section provides examples of signal declarations.

3.2.1 Monitored Variables

Consider binary input signal **mon_HDG_Switch,** which describes the position of a flight control panel button. This is an event-observable signal (see Figure 10). The PROMELA declaration of this signal is presented in Figure 13. PROMELA variable mon_HDG_Switch[current] holds the current value of **mon_HDG_Switch**, and mon_HDG_Switch[previous] holds the previous value of **mon_HDG_Switch**. Figure 13 also shows the declaration of input signal **mon_Indicated_Altitude.** Because it has no events observable in the specification, it is declared as a plain variable holding its current value.

```
#define current 0

#define  previous 1

bit mon_HDG_Switch [2];

short mon_Indicated_Altitude ;
```

Figure 13: Declarations of input signals

3.2.2 Modes and Terms

Consider the flight mode signal **mode_Flight_Director**. It is an internal event-observable signal (see Figure 5). The PROMELA declaration of this signal is presented in Figure 14. mode_Flight_Director[current] holds the current value of **mode_Active_Lateral,** mode_Flight_Director[previous] holds the previous value of **mode_Active_Lateral**, and mode_Flight_Director[new] holds the new value of **mode_Active_Lateral.**

Next, consider mode **mode_Active_Lateral.** It is an internal event-observable signal because event **@F(mode_Active_Lateral=GA)** triggers transition 57, Figure 9. The PROMELA declaration of this signal is presented in Figure 14.

Figure 14 also shows the declaration of **term_Selected_Heading**. Since no event on this signal is observed in the FGS specification, we declare it as a two-element array.

term_Selected_Heading[current] holds the current value of **term_Selected_Heading**; term_Selected_Heading [new'] holds the new value of **term_Selected_Heading.**[1]

```
#define new 2

bit mode_Flight_Director [3];

byte mode_Active_Lateral [3];

#define new' 0

short term_Selected_Heading [2];
```

Figure 14: Declarations of internal signals

3.2.3 Controlled Variables

Figure 16 shows the declaration of controlled variable **con_Vertical_Arm_Text.** This controlled variable is actually an internal event-observable signal because it is used in a sustaining condition, as shown in Figure 15. Figure 16 shows the declaration of controlled variable **con_AP_Engage_Switch_Lamp.** This controlled variable is an output signal, because it is not used elsewhere, so we declare it as a plain variable.

Initial Value:	see value function
Sustaining Condition:	con_Vertical_Arm_Text ≠ ""
Mode	**con_Vertical_Arm_Color**
All Modes	White

*Figure 15: Definition of **con_Vertical_Arm_Color***

```
byte con_Vertical_Arm_Text [3];

short con_AP_Engage_Switch_Lamp;
```

Figure 16: Declarations of controlled variables

3.2.4 Event Identifiers and Event-observable Expressions

Consider the definition of event identifier **@FD_Pressed,** shown in Figure 12. We consider it to be an abbreviation rather than a real signal. We declare **@FD_Pressed,** as shown in Figure 17. We also introduce a new two-element array identifier for the event-observable expression used in the definition of **@FD_Pressed.** The expression expr_FD_Pressed[current] holds the current value of this expression, and expr_FD_Pressed[previous] holds its previous value.

[1] We introduce new' in addition to new because an array array[n] in PROMELA is indexed from 0 to n-1.

```
bool @FD_Pressed;
bool expr_FD_Pressed [2];
```

Figure 17: Declarations of event identifiers and event-observable expressions

3.2.5 Durations

Consider the Autopilot **DISENGAGE** submode transition table, shown in Figure 18. It says that the submode value changes from **Warning** to **Normal** as soon as **Warning** has been continuously true for 10 seconds. We count the duration in terms of simulation cycles. To get the real-time duration, one needs to define the duration of one simulation cycle. To measure the duration, we introduce an auxiliary variable to hold the duration of a particular state. For example, we introduce variable duration_Autopilot_DISENGAGE_Warning, as shown in Figure 19. Initially, we set it to 0. We increment it each simulation cycle, provided the value of Autopilot **DISENGAGE** submode is **Warning.** We reset it to 0 as soon as Autopilot **DISENGAGE** submode leaves this value.

Id	From	Events	To
11	Warning	@T(Duration(INMODE(Warning)) > 10 sec)	Normal

*Figure 18: Autopilot **DISENGAGE** submode transition table*

```
int duration_Autopilot_DISENGAGE_Warning;
```

Figure 19: Declaration of a duration variable

3.2.6 General Structure

The general structure of signal declarations is shown in Figure 20.

```
                                    /* Monitored variables (see Section 3.2.1) */
... < Contents of Figure 13 >; ...
... < Contents of Figure 22 >; ...
                                     /* Modes and terms (see Section 3.2.2) */
... < Contents of Figure 14 >; ...
                                    /* Controlled variables (see Section 3.2.3) */
... < Contents of Figure 16 >; ...
                  /* Event identifiers and event-observable expressions (see Section 3.2.4) */
... < Contents of Figure 17>; ...
                                          /* Durations (see Section 3.2.5) */
... < Contents of Figure 19>; ...
```

Figure 20: General structure of signal declarations

3.3 Signal Initialization

We assume that there are no mutual dependencies between the initial values of different internal signals. CoRE's signal initialization procedure first assigns random initial values to input signals. Then it iteratively initializes those internal signals whose initial value expressions have become defined as a result of the previous initializations. In the absence of mutual dependencies between the initial values of different internal signals, this iteration terminates so that all the internal

signals are initialized.[2] After that, output signals are initialized depending on their sustaining conditions.

Although there are no mutual dependencies between the initial values of different internal signals in the FGS specification, the initialization procedure described above does not directly work. The problem is that sustaining conditions are imposed on some internal signals, and such signals can therefore be undefined. Note that in CoRE internal signals are always defined (once they are initialized). Thus we have to consider the semantics of an undefined value of an internal signal, and how to calculate CoRE expressions over undefined values. In CoRE this is not a problem, because only output variables can be undefined, and output variables are not used in expressions.

To represent the *undefined value of a signal* we use a value out of the signal's range. This very simple interpretation of undefined values works for the FGS specification. The main technical problem with undefined values is the evaluation of expressions over partially defined signals. The only initially undefined signals we found (by applying the procedure above) were some output signals and flight modes. The output signals are not used in expressions elsewhere in the specification. An out-of-the-range value of an output signal can be easily recognized as an exception value. The FGS modes are signals over finite domains. The only atomic expressions in which modes occur are of the form *mode* = *const* or *mode* $\neq$ *const*, where *mode* is a mode, and *const* is a constant in the mode's range. In this context, considering the undefined value of *mode* as a value out of the signal's range is appropriate. However, if other atomic expressions over *mode* were allowed, it would be problematic to evaluate the expressions over undefined values. For example, if the domain of *mode* is ordered, what is the value of the expression *mode* > *const* or *mode* $\leq$ *const* when *mode* is undefined?

The initial values of the internal and output signals in the PROMELA code are assigned according to the FGS specification at the beginning of the target PROMELA process. The signals with constant initial values and with sustaining conditions set to "none" are initialized first. Next, we assign random initial values to input signals.[3] Then we initialize the signals whose sustaining conditions become defined[4] and valid, and whose initial value expressions become defined as a result of the previous initializations. We repeat this step until no more signals can be initialized. In the FGS specification, no initial values of the sustaining conditions depend on the initial values of the input variables. This results in simple detection of the initially undefined signals, and simple analytical dependence of the initial values of the "initializable" signals on the initial values of input signals. The signals that are not initialized at the beginning of the simulation are assigned undefined values. The rest of this section provides examples of signal initialization.

3.3.1 Input Signals

We initialize input signals with random values that are within their range. Since PROMELA does not have such a random number construct, we define macros random1 and random2, Figure 21, based on the binary representation of natural numbers. Given a signal x over an integer range $[A..B]$, expression random1(x,A,B) assigns x randomly so that $A \leq x \leq B$.

[2] Note that in CoRE no sustaining conditions are imposed on internal signals.

[3] For the signals represented by arrays, we initialize all the array elements to be the same value.

[4] We consider an expression defined as long as all its arguments are defined.

```
int range;
bit digit;
#define random1(x,A,B)          \
range = 1;                      \
x = 0;                          \
do                              \
:: if                           \
  :: range > B – A -> break;    \
  :: else -> if                 \
          :: digit = 0;         \
          :: digit = 1;         \
          fi;                   \
          x = 2*x + digit;      \
          range = 2*range;      \
od;                             \
if                              \
:: x <= B-A -> x = A + x;       \
:: else -> x = A + x – (B–A);   \
fi

#define random2(x,A,B)          \
random1(x[current],A,B);        \
x[previous] = x[current]
```

Figure 21: Random number generation

For example, the range of **mon_Indicated_Altitude** is [-8000..56000]. This signal has no observable events. Therefore, we initialize it with a random value within its range, as shown in Figure 22. Now consider binary input signal **mon_HDG_Switch**. This is an event-observable signal. We initialize it using macro random2, as also shown in Figure 22.

```
random1(mon_Indicated_Altitude,-8000,56000);
random2(mon_HDG_Switch, 0, 1)
```

Figure 22: Input signal initialization

3.3.2 Initially Defined Signals

Consider the definition of **con_AP_Engage_Switch_Lamp**, shown in Figure 2. **con_AP_Engage_Switch_Lamp** is initialized by default as shown in Figure 6. According to these figures, the initial value of **con_AP_Engage_Switch_Lamp** depends on the initial value of **term_AP_Engaged**, which is defined as follows:

term_AP_Engaged: boolean ≡ mode_Autopilot = ENGAGED.

The initial value of **mode_Autopilot** is **DISENGAGED** with no sustaining condition. Thus the initial value of **term_AP_Engaged** is **FALSE**, and, according to Figure 2, the initial value of **con_AP_Engage_Switch_Lamp** is **Unlit**. According to the declaration of **con_AP_Engage_Switch_Lamp,** shown in Figure 16, we initialize it using macro init1 as shown in Figure 23.

Consider signal **term_Selected_Heading**. Its initial value is **mon_Heading** with no sustaining condition, and therefore, is a function of an input signal. According to the declaration of **term_Selected_Heading**, Figure 14, we initialize it using macro init2' as shown in Figure 23.[5]

```
#define init1(x,y)          x = y

#define init2'(x,y)         x[current] = y; x[new'] =y

#define init3(x,y)          x[previous] = y; x[current] = y; x[new] =y

init1(con_AP_Engage_Switch_Lamp, Unlit);

init2'(term_Selected_Heading, mon_Heading);
```

Figure 23: Initializing non-input signals

3.3.3 Initially Undefined Signals

Consider **mode_Active_Lateral**. According to Figure 5, its initial value is undefined, since the initial value of **mode_Flight_Director** is **OFF** with no sustaining condition. The possible values of **mode_Active_Lateral** are **ROLL**, **HDG**, **NAV**, **APPR**, and **GA**. In PROMELA, we define the corresponding constants along with "undefined" constant BYTE_UNDEF as shown in Figure 24.[6] According to the declaration of **mode_Active_Lateral**, Figure 14, we initialize it using macro init3 as shown in Figure 24.

```
#define ROLL          0
#define HDG           1
#define NAV           2
#define APPR          3
#define GA            4
#define BYTE_UNDEF  255

init3(mode_Active_Lateral, BYTE_UNDEF);
```

Figure 24: Initializing signals with undefined value

3.3.4 General Structure

The general structure of signal initializations is shown in Figure 25.

---

[5] Since an array identifier *x* is equivalent to *x*[0] in a PROMELA value expression, we use mon_Heading rather than mon_Heading[current] to improve readability.

[6] Considering an "undefined" value may result in increasing the size of the signal declaration type and, consequently, the system state size. However, the effectiveness of the *supertrace* model checking algorithm [6] that we use does not depend on the system state size itself, but rather on the number of reachable states.

```
/* Input signals (see Section 3.3.1) */
... < Contents of Figure 22 >; ...
/* Initially defined signals (see Section 3.3.2) */
... < Contents of Figure 23>; ...
/* Initially undefined signals (see Section 3.3.3) */
... < Contents of Figure 24>; ...
```

Figure 25: General structure of signal initializations

3.4 Expression Translation

3.4.1 Event Expressions

The translation of CoRE events over a binary signal x is presented in Figure 26. We use macros to facilitate the translation.

CoRE Event	*PROMELA Translation*
@CHANGED(x)	#define current 0 #define previous 1 #define at_CHANGED(x) x[current]!= x[previous]
@T(x)	#define at_T(x) (!x[previous]) && x[current]
@F(x)	#define at_F(x) x[previous] && (!x[current])

Figure 26: Translating events

To translate events over non-variable CoRE expressions, we first translate them into the equivalent events over binary signals. The translation of the events over expressions with one binary argument is presented in Figure 27.

CoRE Event	*CoRE Equivalent*
@CHANGED(x = ON)	@CHANGED(x)
@CHANGED(x = OFF)	@CHANGED(x)
@T(x = ON)	@T(x)
@F(x = ON)	@F(x)
@T(x = OFF)	@F(x)
@F(x = OFF)	@T(x)

Figure 27: Equivalent events

To translate CoRE event expressions other than above, we introduce auxiliary PROMELA variables, as discussed in Section 3.2. Consider an event definition or transition table, which we denote as *CONTEXT*, containing an event expression *@X(expr)*. We declare a new PROMELA variable *comp_expr* as a binary two-element array, as illustrated in Section 3.2.4. We translate the event expression *@X(expr)* as though it were an event expression *@X(comp_expr)* over binary

signal *comp_expr*. We also precede the translation of *CONTEXT* with updating *comp_expr* with the translation of *expr* (see Section 4.2.1).

For example, consider the definition of **@FD_Pressed,** shown in Figure 12. Its translation is shown in Figure 28, where expr_FD_Pressed and at_FD_Pressed are declared as discussed in Section 3.2.4.

```
at_FD_Pressed = @T(expr_FD_Pressed);
```

Figure 28: Translating contexts with complex event expressions

We translate the CoRE event construct ***event* WHEN *expr*** as event && expr, where event and expr are the translations of ***event*** and ***expr*** respectively.

We translate the **CONTINUOUSLY *expr*** construct used by the FGS specification designers, which has no semantics in CoRE, as expr, where expr is the translation of ***expr***.

3.4.2 Duration Expressions

We replace DURATION expressions with auxiliary duration variables (see Section 3.2.5). For example, we translate the expression **Duration(INMODE(Warning)) > 10 sec** (shown in Figure 18) as duration_Autopilot_DISENGAGE_Warning > 10. (Here we assume that the duration of one simulation cycle is 1 second.)

3.4.3 General Expressions

The translation of non-event, "value" expressions to PROMELA is straightforward. We replace a signal identifier ***x*** with *x*[current].[7] We replace CoRE built-in operators with their PROMELA analogues, e.g., **AND** with &&, and **OR** with ||. We translate event sub-expressions as described in Section 3.4.1.

3.5 Table Translation

We translate CoRE function tables using PROMELA non-deterministic if-statements. Consider a generic function table presented in Figure 29.

$mode_1$	...	**$mode_n$**	**condition/event**	**result**
val_{11}	...	val_{n1}	$cond_1$	expr
...	...	...	...	
val_{1k}	...	val_{nk}	$cond_k$	$expr_1$

Figure 29: Generic CoRE function table

For a selector table, the **condition/event** column is omitted For a condition table, all expressions in the **condition/event** column are non-event, "value" expressions. For an event table, all expressions in the **condition/event** column are event expressions. For a mode transition table, **n**=1, and **result= mode_1.**

[7] Since an array identifier *x* is equivalent to *x*[0] in a PROMELA value expression, we actually replace *x* with *x* rather than *x*[current] to improve readability.

The translation of the generic table shown in Figure 30 is built from the translations of the table's expressions as discussed in Section 3.4. The variable part of the translation, *result,* depends on the nature of the signal "**result.**" We substitute result[new] for *result* if **result** is an internal signal, and we substitute result otherwise.

```
If
:: mode_1 == val_1_1 && ... && mode_n == val_n_1
                        && cond_1 -> result = expr_1
...
:: mode_1 == val_1_k && ... && mode_n == val_n_k
                        && cond_k -> result = expr_k
fi
```

Figure 30: Translation of the generic CoRE function table

4 Translating FGS Specification to PROMELA: Simulation Cycle

As discussed in Section 3.1, we translate the distributed FGS specification into a single initial PROMELA process, called the *Flight Guidance System Machine* (FGSM). FGSM is a loop that tracks the signal values, where each iteration of the loop corresponds to a tick of real time. FGSM consists of macro definitions, signal declarations, signal initialization and a simulation loop. A pass of the FGSM simulation loop consists of four consecutive parts. The first part, FGSM_I, generates new values for input signals. The second part, FGSM_II, updates the internal signals according to the input events generated by FGSM_II. FGSM_II is an internal loop that implements event cascading in the same way as RSML and VHDL. The internal iteration terminates when all internal events invoked by one-time input events have been processed; but termination is not guaranteed. The third part, FGSM_III, counts the duration the system stays in particular states. The fourth part, FGSM_IV, calculates the values of output signals corresponding to the current values of input and internal signals.

4.1 FGSM_I

In this section, we describe the general format of FGSM_I, the first part of the FGSM loop. FGSM_I updates the current values of the input signals,[8] which generates external events that trigger the iterative computation of new values for internal signals. (According to [1], we may assume that different inputs do not change at the same time.) This is accomplished by making FGSM_I a non-deterministic if-statement that either changes a signal or leaves all signals unchanged. For example, consider input signal **mon_Indicated_Altitude,** described in Section 3.3.1. The branches of the if-statement generating a new value for **mon_Indicated_Altitude** are shown in Figure 31. This method simulates the continuity of the original physical signal. FGSM_I also resets the value of the event cascading flag stable2, as explained in Section 4.2.3. The general format of FGSM_I is shown in Figure 31.

```
if
:: skip;
...<Branches for other input signals>; ...
:: mon_Indicated_Altitude[current] > -8000 -> mon_Indicated_Altitude[current]--;
:: mon_Indicated_Altitude[current] < 56000 -> mon_Indicated_Altitude[current]++;
...<Branches for other input signals>; ...
fi;
/* Initializing stable2 */
stable2 = FALSE;
```

Figure 31: General format of FGSM_I

4.2 FGSM_II

In this section, we describe the general format of FGSM_II, the most complex part of the FGSM loop. FGSM_II simulates event cascading. FGSM_II is an internal cycle consisting of six consecutive parts with the following functionality:

[8] The previous values of input signal are updated by FGSM_II.

1. Update auxiliary variables corresponding to complex event-observable expressions (see Section 3.4.1).

2. Check whether any auxiliary variable changes its value.

3. Check whether any internal event has been generated in the previous iteration (see Items 2, 5); and quit the cycle if no such events are observed.

4. Calculate the new values for the internal signals and event identifiers.

5. Check whether any internal signal changes its value.

6. Update internal signals and input signals.

The iterative updating of internal signals by FGSM_II corresponds to the causal sequencing of simultaneous events. An event cascading mechanism depends on the order of internal signal updating. In our implementation, we first consider the internal events immediately generated by an external event; we then consider the internal events immediately generated by these events; and so on. This kind of signal cascading is also implemented in CSML, VHDL, and Verilog. One may consider a different order of event cascading as in, e.g., Statecharts, [13] to get, in general, different semantics.

We discuss the parts of FGSM_II in the rest of the section.

4.2.1 Updating Complex Event-observable Expressions

As noted, we consider the identifiers of complex event-observable expressions as shorthands. Their values are based on the current values of "real" signals and should be computed before being used elsewhere. These values are used first to check for the absence of internal events (see Sections 4.2.2, 4.2.3). Figure 32 shows the updating of the expression identifier expr_FD_Pressed (see Section 3.2.4, 3.4.1). The order of updating different complex expressions is irrelevant.

```
#update2(x)        x[previous] = x[current]
update2(expr_FD_Pressed);
expr_FD_Pressed[current] = (mon_FD_Switch_left = ON || mon_FD_Switch_right = ON);
```

Figure 32: Updating a complex event-observable expression

4.2.2 Checking on the Changes of Complex Event-observable Expressions

A change of a value of a complex event-observable expression reflects changes of values of "real" signals that occurred in the previous cycle. We introduce boolean variable stable1 to check for the stability of complex event-observable expressions. We calculate its value as shown in Figure 33.

```
#define stable1?(x)      x[current] == x[previous]
stable1 = < conjuncts for the other complex event observable expressions >
    && stable1?(expr_FD_Pressed)
    && < conjuncts for the other complex event-observable expressions >;
```

Figure 33: Checking on the changes of complex event-observable expressions

4.2.3 Checking on the Absence of Internal Events

Internal events consist of those generated by complex event-observable expressions and those generated by internal signals. To check for the absence of the events generated by internal signals, we introduce a Boolean variable stable2. The variable stable2 is assigned FALSE before entering FGSM_II. It is reassigned after computing the new values of internal signals (see Section 4.2.5). We exit the cycle when both stable1 and stable2 become TRUE, as shown in Figure 34.

```
if
:: stable1 == TRUE && stable2 == TRUE -> break;
:: else -> skip;
fi
```

Figure 34: Checking on the absence of internal events

4.2.4 Calculating New Values for Internal Signals and Event Identifiers

4.2.4.1 Event Identifiers

As discussed in Sections 3.2.4 and 3.4.1, we consider event identifiers as abbreviations. The values of event identifiers should be computed before they are used elsewhere. An example of updating an event identifier is shown in Figure 28.

4.2.4.2 Internal Event-observable Signals

Consider signal **mode_Active_Lateral.** Figure 8 represents its mode transition table, and Figure 5 shows its initial value with its sustaining condition. The part of FGSM_II corresponding to the mode transition table is an if-statement presented in Figure 35. The translation is performed as discussed in Section 3.5. We define some additional macros to facilitate the translation. Note that if none of the transitions takes place, skip is executed. The complete functionality of **mode_Active_Lateral**, including the sustaining condition, is shown in Figure 36.

```
#define at_T(x,y)        x[previous] != y && x[current] == y
#define at_F(x,y)        x[previous] == y && x[current] != y
if
/*20*/ :: mode_Active_Lateral == HDG &&
                                        at_HDG_Switch_Pressed->
                                                mode_Active_Lateral[new] = ROLL;
/*21*/ :: mode_Active_Lateral == NAV &&
                                        at_NAV_Switch_Pressed ->
                                                mode_Active_Lateral[new] = ROLL;
/*22*/ :: mode_Active_Lateral == NAV &&
                                        at_NAV_Source_Change ->
                                                mode_Active_Lateral[new] = ROLL;
/*23*/ :: mode_Active_Lateral == APPR &&
                                        at_APPR_Switch_Pressed ->
                                                mode_Active_Lateral[new] = ROLL;
/*24*/ :: mode_Active_Lateral == APPR &&
                                        at_NAV_Source_Change ->
                                                mode_Active_Lateral[new] = ROLL;
 /*25*/ :: mode_Active_Lateral == GA &&
                                        at_T(term_AP_Engaged) ->
                                                mode_Active_Lateral[new] = ROLL;
/*26*/ :: mode_Active_Lateral == GA &&
                                        at_T(term_SYNC) ->
                                                mode_Active_Lateral[new] = ROLL;
/*27*/ :: mode_Active_Lateral == GA &&
                                        at_F(mode_Active_Vertical,GA) ->
                                                mode_Active_Lateral[new] = ROLL;
/*28*/ :: mode_Active_Lateral != HDG &&
                                        at_HDG_Switch_Pressed ->
                                                mode_Active_Lateral[new] = HDG;
/*29*/ :: mode_Active_Lateral != NAV &&
                                        at_NAV_Switch_Pressed ->
                                                mode_Active_Lateral[new] = NAV;
/*30*/ :: mode_Active_Lateral != APPR &&
                                        at_APPR_Switch_Pressed ->
                                                mode_Active_Lateral[new] = APPR;
/*31*/ :: mode_Active_Lateral[current] != GA &&
                                        at_GA_Switch_Pressed ->
                                                mode_Active_Lateral[new] = GA;
    :: else -> skip;
fi
```

Figure 35: Calculating the new value for an internal event-observable signal

```
if
:: mode_Flight_Director == OFF ->  mode_Active_Lateral[new] = BYTE_UNDEF;
:: at_T(mode_Flight_Director)   ->  mode_Active_Lateral[new] = ROLL;
:: else                         ->  <contents of Figure 35>
fi
```

Figure 36: Representing a sustaining condition

4.2.4.3 Internal Signals with No Observable Events

Consider the **term_Selected_Heading** (see Section 3.2.2). Its new value, corresponding to the CoRE definition in Figure 3, is calculated as shown in Figure 37. We represent the *previous value* of the expression **term_Selected_Heading** by the current value term_Selected_Heading, because the notion of "previous" is relative.

```
if
:: @HDG_Knob_Changed -> term_Selected_Heading[new'] =
                                        MOD(term_Selected_Heading+term_HDG_Knob_Rotation,360)
:: else -> skip
fi
```

Figure 37: Calculating new value for an internal signal with no observable events

4.2.5 Checking on the Changes of Internal Signals

To check for the absence of the changes to internal signals, we use variable stable2. FGSM_II assigns stable2, as shown in Figure 38. Unlike the case of complex event-observable expressions (cf. Section 4.2.2), here we monitor the "real" signal events that happen in the current event cascading cycle. Hence the difference between the definitions of stable2? and stable2'?, Figure 38, and stable1?, Figure 33.

```
#define stable2?(x)        x[new] == x[current]
#define stable2'?(x)       x[new'] == x[current]
stable2 = < conjuncts for the other internal signals >
    && stable2(mode_Active_Lateral)
    && stable2'?(term_Selected_Heading)
    && < conjuncts for the other internal signals >;
```

Figure 38: Checking the stability of the internal signals

4.2.6 Updating Input Signals and Internal Signals

Figure 39 shows examples of updating signals of different types. Because no new values for input signals are generated by FGSM_II, it is sufficient to update the previous values of input signals just at the first iteration. Thus, no external events are observed for the rest of the FGSM_II iterations.

```
#update3(x)        x[previous] = x[current]; x[current] = x[new]
#update2'(x)       x[current] = x[new']
   /* Updating input event-observable signals */
update2(mon_HDG_Switch);
   /* Updating internal event-observable signals */
update3(mode_Active_Lateral);
   /* Updating internal signals with no observable events */
update2'(term_Selected_Heading);
```

Figure 39: Updating input signals and internal signals

4.2.7 General Format of FGSM_II

The general format of FGSM_II iteration is presented in Figure 40.

```
/* Updating complex event-observable expressions (see Section 4.2.1) */
... < Contents of Figure 32 >; ...
/* Checking on the changes of complex event-observable expressions (see Section 4.2.2) */
... < Contents of Figure 33 >; ...
/* Checking on the absence of internal events (see Section 4.2.3) */
... < Contents of Figure 34 >; ...
/* Computing new values for internal signals and event identifiers (see Section 4.2.4) */
... < Contents of Figure 35 >; ...
... < Contents of Figure 37 >; ...
/* Checking the stability of internal signals (see Section 4.2.5) */
... < Contents of Figure 38 >; ...
/* Updating input signals and internal signals (see Section 4.2.6) */
... < Contents of Figure 39 >; ...
```

Figure 40: General format of FGSM_II

4.3 FGSM_III

In this section, we describe the third part of FGSM simulation cycle, FGSM_III, which updates the duration variables that store how long the system has stayed in particular states. For example, consider Figure 18, representing the Autopilot **DISENGAGE** submode transition table. The corresponding duration variable duration_Autopilot_DISENGAGE_Warning is updated as shown in Figure 41. Figure 41 also represents the general format of FGSM_III.

```
...<if-statements for other duration variables>;...
if
:: mode_Autopilot_DISENGAGE == Warning -> duration_Autopilot_DISENGAGE_Warning++
:: else -> duration_Autopilot_DISENGAGE_Warning = 0
fi;
...<if-statements for other duration variables>;...
```

Figure 41: General format of FGSM_III

4.4 FGSM_IV

In this section, we describe the general format of FGSM_IV, the fourth part of the FGSM loop. It updates the output signals. Consider **con_Vertical_Arm_Color,** shown in Figure 15. The part of FGSM_IV generating a new value for this variable is shown in Figure 42, where SHORT_UNDEF is defined outside the color range of **con_Vertical_Arm_Color**. FGSM_IV is the sequence of such statements for all the output signals. The order of the statements is irrelevant.

```
...<if-statements for other output signals>;...
if
:: con_Vertical_Arm_Text[current] == EMPTY_STRING ->
                    con_Vertical_Arm_Color = SHORT_UNDEF;

:: else ->          con_Vertical_Arm_Color = White;
fi
...<if-statements for other output signals>;...
```

Figure 42: General format of FGSM_IV

4.5 FGSM: General Format

The general format of FGSM is shown in Figure 43. The d-step statements are inserted to facilitate model checking, as explained in Chapter 7. Figure 43 also includes some code related to the validation of the system properties as explained in Chapter 5.

```
Init
{
d_step{
                        /* Signal declaration (see Section 3.2) */
< Contents of Figure 20>;
                        /* Signal initialization (see Section 3.3) */
< Contents of Figure 25>;
};
/* end d_step */
do ::
            /* Checking on the absence of stuttering (see Section 5.5) */
skip; progress:
                    /*  Updating input signals (see Section 4.1) */
< Contents of Figure 31>;
d_step{
                    /* Updating internal signals (see Section 4.2) */
        do ::
         < Contents of Figure 40>;
        od ;
                    /* Updating durations (see Section 4.3) */
< Contents of Figure 41>;
                   /* Updating output signals (see Section 4.4) */
< Contents of Figure 42 >;
                    /* Checking invariants (see Section 5.3) */
...< Contents of Figure 46>; ...
};
/* end d_step */
od
}
```

Figure 43: General format of FGSM

5 Formalizing the Required Properties

5.1 Completeness

The completeness of a CoRE specification is determined by the completeness of its selector and condition tables. We implement selector and condition tables as PROMELA if-statements, so that the completeness of a selector/condition table corresponds to the absence of deadlock. For example, **con_AP_Engage_Switch_Lamp** is defined as shown in Figure 44. If this definition were incomplete, a deadlock would occur because there is no else branch in the if-statement.

```
If
:: ! term_AP_Engaged -> con_AP_Engage_Switch_Lamp = Unlit
:: term_AP_Engaged -> con_AP_Engage_Switch_Lamp = Lit
fi
```

Figure 44: Representation of a condition table

To check for deadlocks we use the additional run-time option –q in SPIN verification.

5.2 Consistency

Consistency of a CoRE specification is determined by the consistency of its event and condition tables. Consider the event table shown in Figure 44. To check its consistency, we introduce counter COUNT_term_AP_Engaged, set to 0 before control is passed to the table. We also precede the table with the sequence of if-statements incrementing COUNT_ term_AP_Engaged if a transition of the table can be triggered as shown in Figure 45. Then the consistency of the table is expressed by the statement that COUNT_Active_Lateral <= 1.

```
COUNT_term_AP_Engaged = 0;
if
:: ! term_AP_Engaged -> COUNT_term_AP_Engaged ++;
:: else -> skip;
fi;
if
:: term_AP_Engaged -> COUNT_term_AP_Engaged ++;
:: else -> skip;
fi;
assert(COUNT_term_AP_Engages <= 1);
```

Figure 45: Checking on consistency of a condition table

We add such counters for each event or condition table.

5.3 Invariants

Invariants are translated into assert statements at the end of the FGSM cycle (see Figure 43). For example, invariant INV-1, shown in Figure 7, is translated as shown in Figure 46.

```
assert( mode_Active_Lateral != GA || mode_Autopilot = DISENGAGED)
```

Figure 46: Invariant translation

5.4 Unreachable Transitions

The SPIN model checker reports unreachable transitions as unreachable lines of code.

5.5 Stuttering

Stuttering prevents real-time system implementation. In our PROMELA model, stuttering corresponds to infinite iteration of the internal FGSM_II cycle. To check for the absence of stuttering, we put a progress label at the beginning of the external FGSM cycle (see Figure 43). A progress label indicates that this control point should be reached infinitely often. An infinite iteration of the internal FGSM_II cycle would contradict this requirement.

6 State Space Reduction

The huge state space of the FGS specification makes a direct validation by model checking impossible. State explosion is a common model-checking problem [6,14]. The state space reduction techniques that we used to make the model checking feasible are

1. The supertrace algorithm for state exploration,
2. Multiple hashing,
3. Introducing d_step statements, and
4. Input signal abstractions.

Our translation is already optimal because it is a single PROMELA process, which makes reduction techniques related to process interleaving, such as partial order reduction, unnecessary [6]. The state reduction techniques for eliminating irrelevant entries [14] do not work for the FGS specification, because the flight modes are mutually dependent. In the rest of this section we describe the techniques we did use.

6.1.1 The Supertrace Algorithm

The model-checking strategy of SPIN requires it to generate the set of all reachable states. The states are accumulated in accordance with the control flow of the analyzed program. The main problem is to determine whether a current state has been already reached. The supertrace algorithm [6] uses random number coding of states. The generated numbers are stored in a hash table. When a state is analyzed, its number is first compared to those already stored in the hash table. If the number is new, so is the state; and the state's properties are analyzed. Otherwise, the state is considered already analyzed and another search branch is chosen by backtracking. As two different states may have the same encoding, there is a chance that some states may be not analyzed. Thus, the supertrace method is not exhaustive—it can find errors, but not guarantee correctness. Let S be the number of the analyzed states, and N be the number of possible state encodings determined by the size of the hash table. As $S/N \rightarrow 0$ the supertrace method approaches fully exhaustive search.

6.1.2 Multiple Hashing

A hash function that generates state numbers is a parameter of the supertrace algorithm. In the partial supertrace search, different hash functions will result in coverage of different parts of the state space. Therefore, running the supertrace algorithm with different hash functions increases the state space coverage. We ran the supertrace algorithm with 32 built-in hash functions.

6.1.3 Using d_step Statements

The d_step statement introduces a deterministic sequence of code that is executed indivisibly [8]. No states are saved, restored, or checked within a d_step sequence. Therefore, the spurious state explosion due to control flow over deterministic code is eliminated. The only non-deterministic

part of FGSM is supposed to be the FGSM_I, which updates the values of input signals.[9] We insert the d_step statements accordingly (see Figure 43).

6.1.4 Input Variable Abstraction

We reduce the number of reachable states by decreasing the ranges of inputs while preserving the validity of the verified properties. Consider, for example, input signal **mon_VS_Pitch_Count**. Its range is [0..255]. However, only the event **@CHANGED(mon_VS_Pitch_Count)** is used to control flight modes; the actual value of **mon_VS_Pitch_Count** is used just to control output variables. Since we are not interested in the properties of the output variables, we can treat **mon_VS_Pitch_Count** as a binary input signal. Some of the input signals, like **mon_Indicated_Altitude,** are irrelevant to the mode logic, and can be removed from the specification altogether. Proceeding in this way, we modified 12 out of 30 input signals as shown in Figure 49.

Abstracting **mon_Indicated_Airspeed** (shown in Figure 49, Item 2) is a bit different. The only place that **mon_Indicated_Airspeed** influences the flight mode values is in the **mode_Overspeed** transition table, shown in Figure 47. The expression **term_Vmo** is actually an aircraft specific constant. Assuming that 0 ≤ **term_Vmo** ≤ 512, the range of **mon_Indicated_Airspeed** breaks into three intervals: [0.. **term_Vmo**], [**term_Vmo**.. **term_Vmo** + 10], and [**term_Vmo** +10..512]. Within each interval, the actual value of **mon_Indicated_Airspeed** is irrelevant. Therefore, we reduce the range of **mon_Indicated_Airspeed** to [0..2], a value for each interval, and modify the **mode_Overspeed** transition table, as shown in Figure 48

Id	From	Events	To
1	SPEED_OK	@T(mon_Indicated_Airspeed > (term_Vmo +10) AND NOT term_Above_Transition_Altitude)	TOO_FAST
2	SPEED_OK	@T(mon_Indicated_Mach_Number > (term_Mmo + 0.03) AND term_Above_Transition_Altitude)	TOO_FAST
3	TOO_FAST	@T(mon_Indicated_Airspeed ≤ term_Vmo AND NOT term_Above_Transition_Altitude)	SPEED_OK
4	TOO_FAST	@T(mon_Indicated_Mach_Number ≤ term_Mmo AND term_Above_Transition_Altitude)	SPEED_OK

*Figure 47: **mode_Overspeed** transition table*

Id	From	Events	To
1	SPEED_OK	@T(mon_Indicated_Airspeed = 2 AND NOT term_Above_Transition_Altitude)	TOO_FAST
2	SPEED_OK	@T(mon_Indicated_Mach_Number > (term_Mmo + 0.03) AND term_Above_Transition_Altitude)	TOO_FAST

[9] We check on the determinism of the other code, as discussed in Section 5.2.

3	TOO_FAST	@T(mon_Indicated_Airspeed = 0 AND NOT term_Above_Transition_Altitude)	SPEED_OK
4	TOO_FAST	@T(mon_Indicated_Mach_Number ≤ term_Mmo AND term_Above_Transition_Altitude)	SPEED_OK

*Figure 48: Modified **mode_Overspeed** transition table*

Removing the irrelevant input signals from the specification is considered in [14]. However, reducing input signal ranges is not considered there.

	Signal	Original range	Modified range
1	mon_VS_Pitch_Count	[0..255]	[0..1]
2	mon_Indicated_Airspeed	[0..512]	[0..2]
3	mon_Indicated_Altitude	[-8000..56000]	Removed
4	mon_Pressure_Altitude	[-8000..56000]	[0..1]
5	mon_Roll_Angle	[-180..180]	[0..1]
6	mon_Pitch_Angle	[-90..90]	Removed
7	mon_Vertical_Speed	[-32.8..32.7]	Removed
8	mon_Heading	[0..359]	Removed
9	mon_Nav_Source_Frequency_VNR	[108..136]	[0..1]
10	mon_HDG_Count	[0..255]	Removed
11	mon_Speed_Count	[0..255]	Removed

Figure 49: Input signal range modifications

7 Validation Results

Since the supertrace algorithm is a partial search algorithm, the absence of errors in the supertrace validation does not necessarily indicate the absence of errors in the system. On the other hand, errors found with the supertrace algorithm are definitely present. Our analysis found specification errors of different severity: typos, unreachable transitions, and invariant violations.

7.1.1 Typos

Two typos detected by the PROMELA syntactic analyzer were undeclared variables (Figure 50).[10]

Item	Typo	Correction
Section A.9.2.2.1	@NAV_Source_Changed	@Nav_Source_Changed
INV-4	term_On_Ground	mon_On_Ground

Figure 50: Typos detected

7.1.2 Unreachable Transitions

Consider the **mode_Active_Vertical/FLC** transition table, shown in Figure 51. Transition 60 was reported as unreachable code for the following reason. According to the definition of **term_Overspeed**, Figure 11, **term_Overspeed** changes just one event-cascading microcycle after **mode_Overspeed** changes. The **mode_Overspeed** transitions are triggered just by input events. Thus, **@T(term_Overspeed)** can be true just at the third microcycle. On the other hand, according to Figure 9, **mode_Active_Vertical** can go to **FLC** either by transition 50 at the first microcycle, or by transition 52 at a microcycle just after one with true **term_Overspeed**. Thus, **@T(mode_Active_Vertical = FLC)** can be true either at the second microcycle, or after the third microcycle. Therefore, condition **@T(mode_Active_Vertical = FLC) AND @T(term_Overspeed)** is never true.

Id	From	Events	To
58	Entered	@T(mode_Active_Vertical = FLC) AND NOT @T(term_Overspeed)	Track
59	Overspeed	@F(term_Overspeed)	Track
60	Entered	@T(mode_Active_Vertical = FLC) AND @T(term_Overspeed)	Overspeed
61	Track	@T(term_Overspeed)	Overspeed

Figure 51: Flight level change submode transition table

A possible solution is to return to the previous version of the FGS specification [15] where **term_Overspeed** is used in Figure 51 instead of **@T(term_Overspeed)**.

[10] The specification has undergone several manual inspections by its developers.

7.1.3 Invariant Violations

7.1.3.1 INV-7

We detected a violation of INV-7:

mode_Active_Lateral = APPR/Track ⇒ term_Selected_Nav_Type ∈ {LOC, FMS}

term_Selected_Nav_Type is defined by the following condition table:

Conditions	term_Selected_Nav_Type
mon_Selected_Nav_Source = FSM<N>	FMS
mon_Selected_Nav_Source = VNR<N> AND mon_Nav_Source_Signal_Type<VNR<N>> = VOR	VOR
mon_Selected_Nav_Source = VNR<N> AND mon_Nav_Source_Signal_Type<VNR<N>> = LOC	LOC

Figure 52: Definition of ***term_Selected_Nav_Type***

The error trace starts from a state where

1. **mode_Active_Lateral = APPR/Track**
2. **mon_Selected_Nav_Source = VNR<N>**
3. **mon_Nav_Source_Signal_Type<VNR<N>> = LOC.**

According to Figure 52, **term_Selected_Nav_Type** I= **LOC** in the state, so INV-7 initially holds. At the next simulation cycle, let **mon_Nav_Source_Signal_Type<VNR<N>>** changes to **VOR**. According to Figure 52, **term_Selected_Nav_Type** change from **LOC** to **VOR**. However, **mode_Active_Lateral** remains the same, while it should not. The error is possibly because the change of **term_Selected_Nav_Type** does not invoke event **@NAV_Source_Change** triggering transition 24, shown in Figure 8.

@NAV_Source_Change is defined as shown in Figure 53. The error is easy to fix by redefining **@NAV_Source_Change** as shown in Figure 54.

```
@Nav_Source_Change: event ≡ @CHANGED(mon_Selected_Nav_Source) OR
(@CHANGED(mon_Nav_Source_Frequency<mon_Selected_Nav_Source>)
                                        WHEN term_Selected_Nav_Type ∈ {VOR, LOC})
```

*Figure 53: Definition of **@Nav_Source_Change***

```
@Nav_Source_Change: event ≡ @CHANGED(mon_Selected_Nav_Source) OR
((@CHANGED(mon_Nav_Source_Frequency<mon_Selected_Nav_Source>)
        OR @CHANGED(mon_Nav_Source_Signal_Type<VNR>))
                                        WHEN term_Selected_Nav_Type ∈ {VOR, LOC})
```

*Figure 54: Modified definition of **@Nav_Source_Change***

7.1.3.2 INV-9

We also detected a violation of **INV-9:**

mode_Altitude_Select = ACTIVE ⇔ mode_Active_Vertical=ALTSEL

The altitude select ENABLED submode transition table is shown in Figure 55.

Id	From	Events	To
64	ARMED	@T(term_ALTSEL_Cond = Capture AND Duration(INMODE) > const_min_armed_period)	ACTIVE
65	ACTIVE	@F(mode_Active_Vertical ∈ {APPR, GA, ALTHOLD, ALTSEL})	ARMED

Figure 55: Altitude select ENABLED submode transition table

The error trace starts from a state where

1. **mode_Altitude_Select = ARMED,**
2. **mode_Active_Vertical = FLC,**
3. The events triggering transition 46, in Figure 9, and transition 64, in Figure 55, occur.

The error trace is shown in Figure 56. The event triggering transition 46 is *input* event **@ALT_Switch_Pressed**. The event triggering transition 64 is *internal* event **@T(term_ALTSEL_Cond = Capture AND Duration(INMODE) > const_min_armed_period).** Therefore, the error trace does not violate the FGS assumption about admissible simultaneous events (see Section 4.1). This assumption is really an assumption of determinism. In the situation above, determinism is maintained, but an invariant is violated. We do not see an obvious way to correct the problem. Several ways to handle simultaneous events are discussed in [1]. In any case, the discussed violation is an inherent feature of the system functionality and should be resolved by the specification designers. We have reported all of the detected errors to the developers of the FGS specification.

mode_Altitude_Select	mode_Active_Vertical
ARMED	**FLC**
Transition 64	Transition 46
ACTIVE/Capture	**ALTHOLD**
Transition 63	Transition 43
CLEARED	**ALTSEL**
Transition 62	None
ARMED	**ALTSEL**
None	None

Figure 56: An error trace

8 Conclusion

8.1 Project Results

The project has achieved the following results:

1) We developed a technique for translating the extended CoRE formalism (including event cascading, continuous events and partially defined internal signals) into PROMELA, the input language of SPIN. The translation is optimal for model checking because

   a) The resulting specification consists of a single PROMELA process, which consequently has no interleaving,

   b) The deterministic part of the target code (all but the generation of the input signal values) can be treated as a d-step, eliminating the state explosion due to the internal control flow. This assumes that the CoRE specification is consistent.

2) Within the PROMELA model we represented certain basic CoRE requirements:

   a) Completeness,

   b) Consistency,

   c) Invariants,

   d) The absence of unreachable transitions, and

   e) The absence of stuttering.

3) We applied the advanced state-space reduction techniques in handling the large state space of the FGS specification, in order to make the model checking feasible:

   a) The supertrace algorithm,

   b) Multiple hashing, and

   c) Input variable abstraction.

4) As a result, we detected several specification errors of different degrees of importance:

   a) Typos,

   b) Unreachable transitions, and

   c) Invariant violations, including an intricate one due to unexpected simultaneous events.

8.2 Directions For Future Work

We propose the following directions for future work:

1) Implement a translator from CoRE to PROMELA, based on the translation techniques we have developed,

2) Develop new methods of state space reduction, and

3) Translate the FGS specification into RSML, Verilog, or VHDL, and validate it using related state exploration methods. An especially interesting possibility is to use symbolic model checking procedures based on binary decision diagrams [16]. In many cases these procedures can exhaustively analyze models with state spaces much bigger than those analyzed by traditional methods.

References

1. Steven P. Miller and Karl F. Hoech. Specifying the Mode Logic of a Flight Guidance System in CoRE, Version 1.1. Technical Report, pp. 109. Collins Commercial Avionics, Rockwell International, June 17, 1997.

2. David Hughes and Michael Dornheim. Automatic Cockpits: Who's in Charge? : Parts I & II. *Aviation Week & Space Technology,* January 30 – February 6, 1995.

3. Stuart R. Faulk, Lisa Finneran, James Kirby, and Assad Moini. Consortium Requirements Engineering Guidebook. Technical Report SPC-920600-CMC, Software Productivity Consortium, Herndon, VA, December, 1993.

4. The VIS Group. VIS: A system for Verification and Synthesis. In the *Proceedings of the 8th International Conference on Computer Aided Verification*, pp. 428-432, Springer, Lecture Notes in Computer Science 1102. Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996

5. FormalCheck™ Home. http://www.bell-labs.com/project/formalcheck/index.html

6. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

7. CV: A Model Checker for VHDL. http://www.cs.cmu.edu/~modelcheck/cv/project.html

8. On-The-Fly, LTL Model Checking With SPIN. http://www.netlib.no/netlib/spin/whatispin.html

9. Nancy G. Levenson, Mats P.E. Heimdahl, Holly Hildreth, Jon D. Reese. Requirements Specification for Process-Control Systems. In *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684—107, 1984.

10. *IEEE Standard VHDL Language Reference Manual.* IEEE Std 1076-1993, IEEE Standards, 1994.

11. Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

12. Mats P.E. Heimdahl and Nancy Leveson. Completeness and Consistency Analysis of State-Based Requirements. In *IEEE Transactions on Software Engineering*, May 1996.

13. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, pp. 231—274, 1987.

14. Ramewsh Bharadwaj, Constance Heitmeyer. Verifying SCR requirements specifications using state explorations. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

15. Steven P. Miller and Karl F. Hoech. Specifying the mode logic of a flight guidance system in CoRE. Technical Report, pp. 107. Collins Commercial Avionics, Rockwell International, April 4, 1997.

16. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, David L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* vol. 13(4), 1994.

Appendix

A Implementing Event Broadcasting in PROMELA

In this section, we discuss how to formalize triggering-processes-by-event-broadcasting in PROMELA. The discussion suggests that a feasible formalization would require N global variables, where N is the sum of the lengths of the sensitivity lists of the running processes.

Consider a CoRE specification consisting of three classes, A, B, and C, tracking integer signals x, y and z respectively. Let x be a randomly generated input signal, $y=x+1$, and $z= z -1$. A naive formalization of classes A, B and C as concurrent PROMELA processes would look as follows:

```
bit x, y;
proctype A ()
{
do
:: x = 0
:: x = 1
od
}
proctype B ()
{
do
:: y = x-1
od
proctype C ()
{
do
:: z = x+1
od
}
init {run A();run B(); run C()}
```

However, this program is not adequate to the given specification because of the following program trace:

x	0	2	3	1...
y	0	0	2	2...
z	0	3	3	3...

Figure 57: A program trace

This trace shows that processes A, B and C run asynchronously, and sharing domain variable x does not enforce their synchronization. One possible way to enforce the synchronization is to introduce a flow control flags x_in_B and x_in_C corresponding to the sensitivity list of the running copies of B and C as follows:

```
bit x, y;
bool x_in_B, x_in_C;
proctype A ()
{
do
:: x_in_B && x_in_C ->
                    if
                        :: x = 0
                        :: x = 1
                    fi
                    x_in_B = 0;
                    x_in_C = 0;
od
}
proctype B ()
{
do
:: x_in_B -> y = x-1; x_in_B = 0
od
}
proctype C ()
{
do
:: x_in_C -> z = x+1; x_in_C = 0
od
}
init {run A(); run B(); run C()}
```

In general, we introduce one control flag for each signal in the sensitivity list of a process, for each running copy of the process. (This means that we would also have to distinguish the declarations of the running copies of a process.) Using channels for process synchronization seems even less efficient since it would also require additional variables to read from the channels. Even assuming that a final implementation of the CoRE specification has explicit flow control, it is reasonable to avoid it on the early stages of design as a source of additional errors.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1998	Contractor Report

4. TITLE AND SUBTITLE
Flight Guidance System Validation using SPIN

5. FUNDING NUMBERS
522-33-31-01
NAS1-20335

6. AUTHOR(S)
Dimitri Naydich and John Nowakowski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates
Cornell Business & Research Park
33 Thornwood Drive
Ithaca, NY 14850-1250

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/CR-1998-208434

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Ricky W. Butler
Final Report, Task 7

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 59 Distribution: Standard
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

To verify the requirements for the mode control logic of a Flight Guidance System (FGS) we applied SPIN, a widely used sofware package that supports the formal verification of distributed systems. These requirements, collectively called the FGS specification, were developed at Rockwell Avionics & Communications and expressed in terms of the Consortium Requirements Engineering (CoRE) method. The properties to be verified are the invariants formulated in the FGS specification, along with the standard properties of consistency and completeness. The project had two stages. First, the FGS specification and the properties to be verified were reformulated in PROMELA, the input language of SPIN. This involved a semantics issue, as some constructs of the FGS specification do not have well-defined semantics in CoRE. Then we attempted to verify the requirements' properties using the automatic model checking facilities of SPIN. Due to the large size of the state space of the FGS specification an exhaustive state analysis with SPIN turned out to be impossible. So we used the supertrace model checking procedure of SPIN that provides for a partial analysis of the state space. During this process, we found some subtle errors in the FGS specification.

14. SUBJECT TERMS
Formal Methods, Software Verification, Flight Guidance, Model Checking

15. NUMBER OF PAGES
50

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	